CIRCUIT CELLAR ONLINE

FEATURE ARTICLE

Stuart Allman

Implementing a Simple USB Interface

For an Embedded Processor

When connecting modules in your embedded system, decisions need to be made. The universal serial bus (USB) may help you make up your mind and ease the aches of development. With the news that Intel is phasing out the good old RS-232 and replacing it with the USB, Stuart shows us that all is not lost, because the USB should work just as well. He supplies us with all the info and leaves no room for excuses, so get ready to make a game plan for your next design.

Deciding how to connect all of the modules of your embedded system together can be similar to playing a game of "would you rather" with your coworkers. Do you design a new interface or use an industry standard and accept a learning curve? How do you know that an industry standard interface will meet all of your needs before you design it in? What becomes an even worse predicament is, how do you test the interface if you have only your system to connect it to? The universal serial bus (USB) just may be the elixir that takes away some of your development pains.

Yes, it's sad but true, Intel plans to phase out your old trusty RS-232 serial port on the back of your PC in favor of the USB. I, along with many of you, used the old trusty RS-232 serial interface for years. It is incredibly simple, and you have complete control over everything that goes on the bus. So, the question becomes, what do you use now that will be simple, allow complete control, and be immediately understandable?

The answer I have come to in recent years has been the USB. In this article, I'll show how getting up and running on the USB can be painless and almost a direct drop in replacement for RS-232 solutions. I'll show you three example processors that I have developed firmware for and how they can transfer data on the USB.

TYPICAL USB SOLUTIONS

Unlike RS-232, the USB is a master-slave configured bus that must have a central host and can have up to 127 peripherals attached. In other words, only the USB host initiates data transfers, and your device must be willing to share the bandwidth. Currently there are only two speeds available for V.1.1 of the USB, 12 Mbps for full speed and 1.5 Mbps for low speed, with a third (480 Mbps) on the way for USB 2.0. Even with the fastest microcontrollers on the market today, it's impractical to try to bit-bang USB

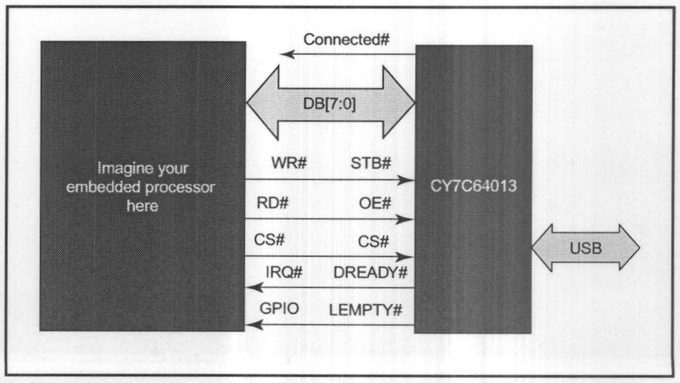


Figure 1—*Here you can see the connection from the embedded processor to the HAPI.*

traffic and remain within the USB timing specifications.

USB peripheral connectivity solutions generally exist in three forms. First is the standalone serial interface engine (SIE). Many vendors sell these so that an embedded processor can set up packets at the transaction level and have the SIE format the packet and send the data at the physical level. An SIE solution has the drawback of making the embedded processor firmware have to understand and communicate at the transaction level. Data cannot be simply transferred without understanding the USB protocol.

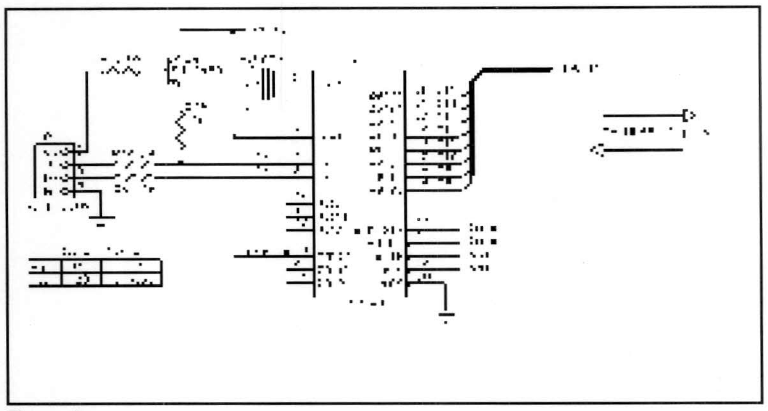
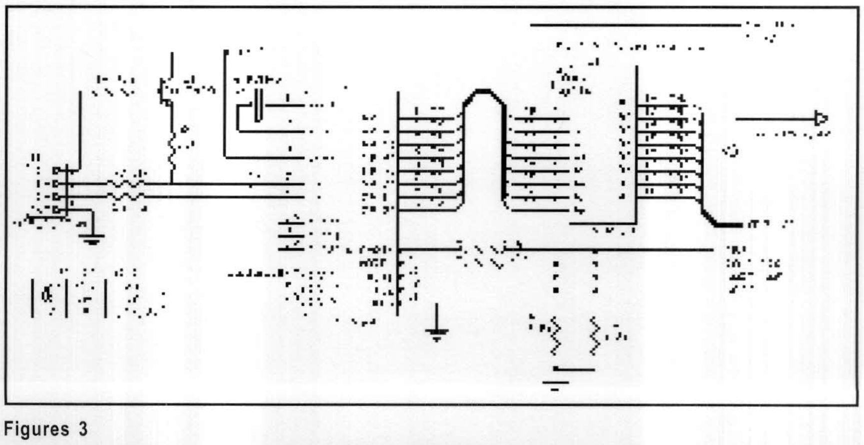


Figure 2

The second common solution is to integrate the SIE into a micro-controller. Literally dozens of silicon vendors provide anywhere from bare-bones to high-power microcontroller solutions for the USB. Typically these are used in peripherals in which a micro-controller can act as the main system controller. This high level of integration allows for cost-competitive solutions in a tiny package, however, the microcontroller firmware still has to understand and interpret data at the USB transaction level.

And, for this article, I will be examining a third solution for systems that already have a central microprocessor and want to easily integrate USB connectivity, similar to RS-232 connectivity. This solution involves letting a separate USB microcontroller act as a peripheral handling the transaction level, freeing the central microprocessor to simply send and receive bytes at the application level. The only drawback of this scheme seems to be a moderate hit to throughput because the microcontroller must handle the transaction level. Still, much larger throughput than RS-232 can easily be achieved.


Figures 3

REQUIRED COMPONENTS

There are five main elements that are required to communicate on the USB—USB SIE, peripheral processor, peripheral firmware, host driver, and host application.

The USB SIE can exist as a standalone memory-mapped peripheral or be integrated in a microcontroller. The SIE takes data at the transaction level and transfers the bits out at the physical level, and vice versa.

The second element is the peripheral processor. Most USB solutions today demand a processor solution of some kind to handle USB transactions, although some USB peripherals are designed to communicate without any processor control. Generally the processor handles the system control features, such as optics and buttons in a USB mouse, as well as the USB transactions.

And, unless you have an intelligent SIE, there is going to have to be firmware somewhere in your peripheral to communicate at the USB transaction level. That's where the peripheral firmware comes in.

The USB is generally supported on three platforms, Windows98/2k and MacOS, with some limited support on Linux. For the purposes of this article, I will be using a Windows WDM driver. The driver recognizes and becomes attached to your device after you plug the device into the host. The peripheral provides the correct Product ID (PID) and Vendor ID (VID) during the startup process.

The host application opens the driver and sends and receives data from the peripheral. USB devices have a logical communications channel known as endpoint 0, which allows you to send vendor-specific commands to the peripheral. Later, I will present a Win32 dialog application that allows you to send data on endpoint 2 (BULK OUT), receive data on endpoint 1 (BULK IN), and send vendor-specific requests on endpoint 0.

AN EASIER SOLUTION

The USB micro-controller I'll discuss is the Cypress Semiconductor CY7C64013. This microcontroller has a bidirectional parallel interface called HAPI (hardware assisted parallel interface) that easily connects to most standard embedded processor buses (see Figure 1).

With a bit of firmware magic in the CY7C64013, all the embedded processor has to do to communicate on the USB is read a byte when it gets interrupted by the HAPI DREADY# signal and write a byte at its leisure

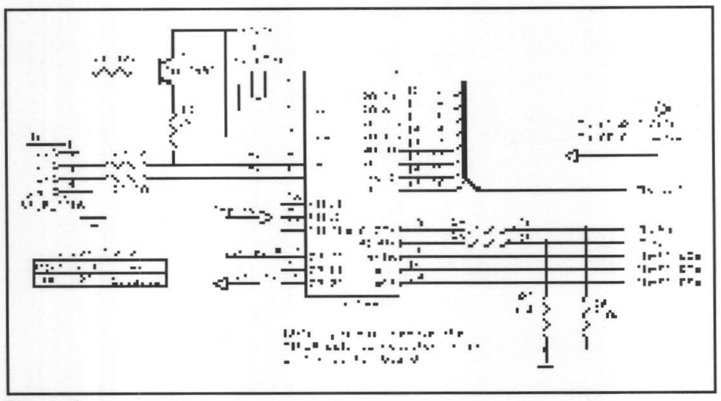


Figure 4

when the LEMPTY# is asserted. Using this interface you should be able to achieve data rates of around 60 to 80 kbps—more than enough bandwidth to run circles around RS-232. This is obviously less than the theoretical maximum throughput of about 1 MBps required by the USB V.1.1 specification, but there are a lot of embedded systems out there that don't require ultra-high speed.

The CY7C64013 microcontroller handles the USB start-up process. This is known as USB enumeration. During this time, the USB host queries the peripheral for data about what the device is and which logical communication channels the peripheral will communicate on. Then the main USB driver on the host connects the peripheral to the device-specific driver using the VID and PID acquired from the peripheral.

An obstacle when using USB is making a device enumerate on the bus and be "Chapter 9" (commonly used to refer to Chapter 9 of the USB specification) compliant. The enumeration process consists mainly of two types of transactions—control read and no-data control—and a list of associated device descriptions transferred during this time, aptly called descriptors.

FOLLOWING THE RULES

There are a number of rules that the device must follow that are not immediately obvious for a peripheral to be USB compliant, and often these cause a great deal of forehead bruising for the first-time developer. If these tasks are taken care of by the CY7C64013 microcontroller, then the embedded processor doesn't even need to know that it's connected to the USB.

Luckily for you (and your sanity), that piece of firmware has already been written and comes with the design package for this article. Enumerating the bus is nearly a standard procedure, so the CY7C64013 firmware can become a black box in your design. All you will need to specify are a few parameters for the enumeration to be technically correct for your particular device. I have specified default parameters to get you up and running, but obviously these will change during your development process. The parameters are defined as constants in the *embedinf.h* file and include VID, PID, power source, and current draw.

Vendor ID is a two-byte value assigned by the USBIF www.usb.org) when you join. I have set this to 0x04B4, the Cypress Semiconductor VID.

Product ID is a two-byte value assigned by you to identify your USB products. This default value of this parameter is arbitrarily set to 0x6401. Assign this parameter however you wish, as long as it doesn't conflict with another product with the same VID.

The power source can be bus- or self-powered, and the host knows if you're going to draw current from it. By default this value is set to self-powered. One of the advantages of USB over RS-232 is that the USB host can provide up to 100 mA for a standard bus-powered device and up to 500 mA for a high-power device. See the USB specification Reference at the end of the article for more details on power limits.

If it is bus-powered, then the host needs to know how much current you plan to draw. By default, the current draw is zero because the device is self-powered.

Besides the physical connections to the USB, this is all you need to know. If you can write to memory without hurting yourself, you can get on the bus.

When you change these parameters, you need to recompile the *embedinf.c* microcontroller firmware using the ByteCraft M8 Series C compiler and program a microcontroller. Unfortunately, the microcontrollers are OTP, so plan on having a few on hand if you intend to test different products using this interface.

FIRMWARE

Typically, the CY7C64013 HAPI will be connected to a microprocessor data bus. For demonstration purposes, I have written firmware for an MC68331, AT91M63200 (ARM7DTMI) processor, and ADSP-21065L DSP to interface to the USB. Connection diagrams for each of these processors are shown in Figures 2, 3, and 4, respectively. The MC68331 code was developed on a proprietary system, the ARM7TDMI code was developed on an AT91EB63 evaluation kit, and the SHARC code was developed using a SHARC 21065L EZ-LAB. Because the processor interface is generic, I won't spend time discussing the specifics of these processors.

The USB driver that comes with the design package for this article is a generic device driver that currently comes with the Cypress EZ-USB development kit. Before you plug in the device, you need to install the *ezusb.sys* driver using the *ezusbw2k.inf* file. In the *ezusbw2k.inf* file, you will find a list of device names and their associated VID and PID. Anytime you modify the VID and PID of your peripheral, you need to reinstall this device driver so the main USB driver can associate your device driver with the peripheral.

Automatic driver creation tools for the Microsoft Windows platform are also available from companies such as Jungo, BSQUARE, and NuMega. Some of these tools are wizard-driven, and others allow you to just plug in your device and the software configures a driver for you. This software can help you avoid the nightmare of developing a WDM driver and will probably equal the cost of hiring a consultant to create a custom driver for you. The driver creation tools will also require that you have a Windows compiler such as Visual C++ along with the Microsoft DDK.

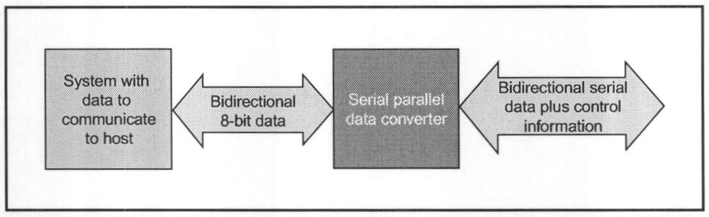

System with data to communicate to host
Bidirectional 8-bit data
Serial parallel data converter
Bidirectional serial data plus control information

Figure 5—*Here you can see the big picture. Note that you can abstract the specific nature of USB and just leave the data-passing interface. This is what your embedded processor should see.*

There is also the possibility that, with a little modification of the USB descriptors in *embedinf.h,* you can use the Microsoft Point of Sale (POSUSB) driver. This driver effectively allows you to interface your USB device like a RS-232 port on your USB host. For more details, check out the Resources at the end of the article.

The drawback of using this driver is that you lose the ability to form your own vendor-specific requests. You also may need to investigate throughput with this driver because it was designed to emulate serial port connectivity to point-of-sale devices.

In order to test your new, fabulous USB interface, I have written a test application that connects to the *ezusb.sys* device driver. The host application is shown in Photo 1 , and a number of vendor-specific requests are shown in Tables 1 and 2 <LINK>. Included are *GET_IO, SET_IO, SET_TIMEOUT,* and *SEND_IMMEDIATE.*

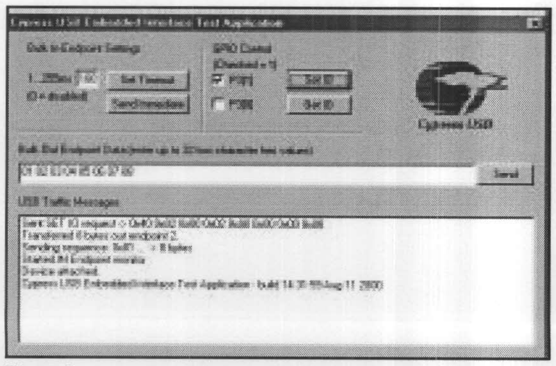
VENDOR-SPECIFIC REQUESTS

GET_IO gets the values on port1[2:0]. These bits are hi-z inputs and should be pulled high or low if the device is bus-powered. If the inputs are left floating while the device is suspended, then the inputs may consume too much current to meet the 500-µA USB suspend current specification.

SET_IO is written to port3[1:0]. These pins have an internal pull-up resistor and CMOS pull down, so you can use these pins as a host-controlled reset, button pull-up resistor, or any other output function.

The IN endpoint communication channel has a buffer size of 32 bytes. SET_TIMEOUT is used to set a transmission timeout value on this buffer. If the timeout value is set to 0 ms (by default, but you can change the default in *embedinf.h*), then the microcontroller will wait for the device to fill up the endpoint buffer before sending data to the USB host. If this value is between 1 and 255 ms and the embedded processor has not written enough bytes to fill up the buffer in that time interval, the microcontroller will time out and send all the bytes in the IN endpoint data buffer. This is meant to handle the situation when the embedded processor may only have 5 bytes to send. If the timeout period is nonzero, then the data will make it to the host after the timeout period, otherwise the data will remain in the IN endpoint buffer until a full 32 bytes are written into the HAPI by the embedded processor.

And finally, *SEND_IMMEDIATE* causes the microcontroller to send whatever data it has in the endpoint 1 buffer to the USB host immediately.


Cypress USB Embedded Interface Test Application
Bulk In Endpoint Settings
1...255ms
(0 = disabled)
Set Timeout
SendImmediate
GPIO Control
(Checked = 1)
P3[1]
P3[0]
Set IO
Get IO
Cypress USB
Bulk Out Endpoint Data (enter up to 32 two character hex values)
Send
USB Traffic Messages

01 02 03 04 05 06 07 08

Sent SET_IO request -> [illegible]
Transferred 8 bytes out endpoint 2.
Sending sequence: 0x01 ... -> 8 bytes
Started IN Endpoint monitor
Device attached.
Cypress USB EmbeddedInterface Test Application - build [illegible] 2000

Photo 1

One thing you have to keep in mind when thinking about USB transfers is that everything is defined to be host-centric. Each device is assigned an address during the enumeration process. After enumeration, the host communicates with the peripheral using logical communication channels that exist at that address. These logical communication channels are called endpoints. Each endpoint is assigned a size, direction, and type of transfer mechanism allowed. A peripheral's OUT endpoint accepts data from the USB host. A peripheral's IN endpoint sends data to the USB host.

The host application allows you to send up to 32 bytes at a time to the embedded processor (OUT endpoint) buffer in the microcontroller. These bytes are then transferred from the OUT endpoint buffer to the central microprocessor via the HAPI. Data transfers from the host to central microprocessor cannot occur again until the central microprocessor reads all of the bytes from the previous transfer.

The host application continuously monitors the IN endpoint while the device is attached to the host. Whenever a packet is sent from the peripheral to the host, the data is displayed in the window at the bottom of the application.

HOST TRANSFER MECHANISMS

Now that I've caught your attention and you are thoroughly salivating with creative thoughts, it's time to think about how you are going to communicate with the host computer. Unless your platform already has a device class supported for your device, you will need to write a custom host application for your new, fabulous gizmo.

The EZ-USB driver used in this example uses the older file name mechanism for opening the driver. Newer drivers that conform to the WDM standard will use a globally unique ID (GUID) to access the driver. In order to open the USB device driver for reading and writing, the host application makes a call to *bOpenDriver()* with the file name specifying the device it wants to open. The system then returns a handle to the device (*bOpenDriver(&hDevice, "Ezusb-0");.*).

The identifier *"Ezusb-0"* is the first USB peripheral plugged in that uses the

Listing 2

```
bulkControl.pipeNum = 0;

bResult = DeviceIoControl ( hDevice,
    IOCTL_EZUSB_BULK_READ,
    &bulkControl,
sizeof(BULK_TRANSFER_CONTROL),
    inbuffer,
    32,
    &numBytesReturned,
    NULL);
```

Listing 3

```
vendor_request.request = 3;
vendor_request.value = 0;
vendor_request.index = 0;
vendor_request.direction = 1;
vendor_request.recepient = 0;
vendor_request.requestType = 2;
vendor_request.requestTypeReservedBits = 0;

bResult = DeviceIoControl (hDevice,
IOCTL_EZUSB_VENDOR_OR_CLASS_REQUEST,
    &vendor_request,
sizeof(VENDOR_OR_CLASS_REQUEST_CONTROL),
    buffer,
    1,
    &numBytesReturned,
    NULL);
```

Listing 4—*need caption.*

```
    vendor_request.request = 2;
    vendor_request.value = portval;
    vendor_request.index = 0;
    vendor_request.direction = 0;
vendor_request.recepient = 0;
vendor_request.requestType = 2;
vendor_request.requestTypeReservedBits = 0;

    bResult = DeviceIoControl ( hDevice,
    IOCTL_EZUSB_VENDOR_OR_CLASS_REQUEST,
        &vendor_request,
    sizeof(VENDOR_OR_CLASS_REQUEST_CONTROL),
        buffer,
        0,
        &numBytesReturned,
        NULL);
```

[illegible] of vendor-specific requests formatted as control read and no-data control transfers. They are defined as follows:

- Bulk out—the computer sends information to the peripheral using a bulk endpoint
- Bulk in—the peripheral sends information to the computer using a bulk endpoint
- Control read—the computer requests specific data on endpoint 0, and the peripheral returns the requested data on endpoint 0
- No-data control—the computer sends a control parameter to the peripheral on endpoint 0. A zero length acknowledgment packet is sent by the peripheral to the computer on endpoint 0.

The bulk out transfers occur by specifying endpoint 2, the transfer mechanism, and data buffer location to a *DeviceIOControl*() function call. Note that the *bulkControl.pipeNum* parameter is zero-based, so you need to write a value of one to this parameter. The rest of the parameters are constants sent to the driver to specify a bulk out transfer, output buffer, and the result of the transaction. When the transfer has completed or if there was an error in transmission, the *DeviceIoControl()* function will return with the result in *iResult* (see Listing 1).

Bulk in transfers are similar, except they specify the constants for a bulk in transaction and a buffer to capture data. Note that the function call specifies to get 32 bytes, which is the endpoint size of the CY7C64013. If the device returns 32 bytes or less, this function will return with the number of bytes captured in *numBytesReturned* (see Listing 2).

Control read transfers are a bit different in that you fill a VENDOR_OR_CLASS_REQUEST_CONTROL structure and specify a vendor-specific request to the *DeviceIoControl()* function call. In Listing 3, the *vendor_request* structure is specifying the parameters to perform a control read for the *GET_IO* request. The request should return one byte in the buffer parameter.

The last transfer type used in the example application is a no-data control transaction used during a *SET_IO* command. The transfer parameters are basically the same as a control-read, only the direction is changed to host-to-device. In this case, no data is returned from the peripheral. The peripheral simply returns a zero length packet to acknowledge that it received the request (see Listing 4).

The other transfer types that are available with this driver are isochronous, interrupt, and control writes. The EZ-USB driver documentation describes how to use these, so I will leave these transfer types for you to explore.

HOW DOES IT WORK?

Now that I have gone into more than enough detail, it's time to step back and get the 10,000¢ view. If you abstract the functionality of the microcontroller, then the view will look something like Figure 5. You have data and control information coming in and out of the USB microcontroller, which appears as a simple data-exchanging device. Data coming from the USB host is passed through the CY7C64013 microcontroller and read by the embedded processor.

Likewise, when the embedded processor writes a byte to the CY7C64013 microcontroller, it is transferred to the USB host. Timing information for the CS#, STB#, OE#, LEMPTY#, and DREADY# signals can be found in the CY7C64013 datasheet (see Resources). Figures 6 and 7 show the relative timing of the HAPI read and write sequences.

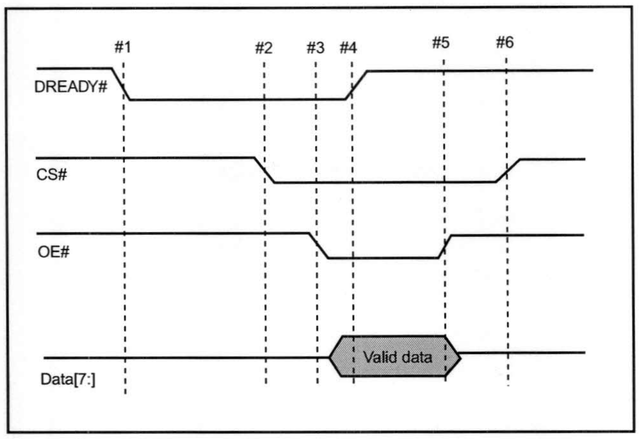


Figure 6

The central microprocessor may optionally read the CONNECTED# signal (port3[2]) from the microcontroller to determine if the data written to the microcontroller's HAPI will be transferred anywhere, but this still does not require any knowledge of the USB. In effect, the central microprocessor does not need to know anything about USB to aptly handle USB; and with an interface similar to a 16550 UART, it's almost a drop in replacement! The protocol that you have already developed for your product can remain in place without modification.

One requirement of my original design enabled the USB host to reset the peripheral. The CY7C64013 already has a power-on reset circuit built in, so the ability to control the pins at port[1:0] allowed me to pick a pin and use it as a POR pin. Because of the built-in 14-kilohm pull-up resistor, you can also use these pins in parallel with open-drain POR circuits that may be in your system already. Need both polarities of POR? No problem, use both pins.

The port1[2:0] pins allow you to attach buttons or status pins from your system and have the USB host read these values.

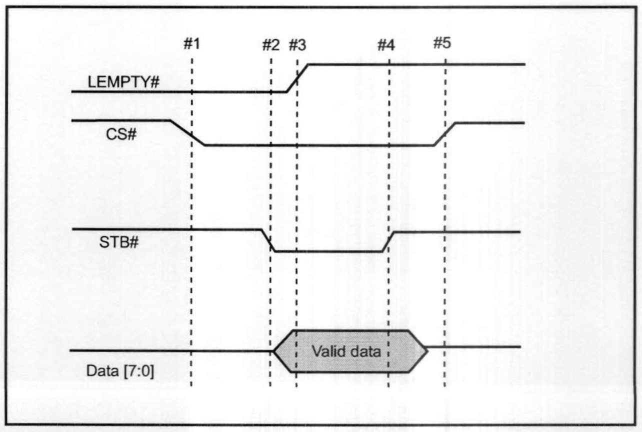


Figure 7

For instance, if you want to wait for the embedded processor to boot before sending any data down the USB pipe, you could monitor a pin on the processor from the host. These pins are in a high-impedance state, so you would need to provide a pull-up resistor for switches or use port3[1:0] as pull-up resistors. Feel free to be creative.

THE GRAND CONCLUSION

The embedded processor doesn't have to know that you've taken it off the old bus and put it onto a new one. This frees your embedded processor to handle your main system tasks and leave USB for the background. Plus, you can keep your same protocol if you are currently using RS-232.

Even moderate data throughputs for USB can make RS-232 solutions look old and tired. It's time to accept the fact that USB is here to stay and you may have to abandon your beloved RS-232 port.

The grand conclusion that I was trying to persuade you to come to is that the aspects of the USB can be hidden by the simple use of a USB microcontroller. If you treat the CY7C64013 microcontroller as a black box with data coming in and out of either end, then the design of an embedded system using USB becomes much simpler. I've given you the microcontroller firmware, USB host driver, USB host example and test application, and example firmware for multiple embedded processors. What excuse could you possibly have for not giving it a try in your next design?

Stuart Allman works for Cypress Semiconductor in the USB microcontroller development tools group. In his spare time, he is a DSP guru wannabe and is always up for a good audio discussion. He can be reached at sea@cypress.com.

SOURCES

Embedded USB interface design package and CY7C64013
Cypress Semiconductor Corp.
(408) 943-2600
Fax: (408) 943-6848
www.cypress.com/design/progprods/usb/usbrefdesign.html

MC68331
Motorola Semiconductor Products
(602) 952-4103
Fax: (602) 952-4067
www.mot-sps.com

AT91M63/4399 (ARM7DTMI)
Atmel Corp.
(408) 441-0311
Fax: (408) 436-4200
www.atmel.com

ADSP-21065L DSP and SHARC 21065L EZ-LAB
Analog Devices, Inc.
(617) 329-4700
Fax: (617) 329-1241
www.analogdevices.com

REFERENCES

Cypress Semiconductor Corp., "CY7C64013 CY7C64113 Full-Speed USB (12 Mbps) Function," 2000.

USB Revision 2.0 Specification, www.usb.org/developers/docs.html.

USB development tools, www.usb.org/developers/tools.html.

Microsoft Windows Point of Sale (POSUSB) driver information, www.eu.microsoft.com/hwdev/usb/posusb.htm.

RESOURCES

Jungo USB driver development tools information, www.jungo.com/windriver.html.

BSQUARE USB driver development tools information, www.bsquare.com.

Compuware Corp., NuMega USB driver development tools information, www.numega.com/drivercentral/default.asp.

Linux USB project information, www.linux-usb.org.

Apple Computer, Inc., Macintosh USB hardware and driver information, http://developer.apple.com/hardware/usb/.